BOOK 3: REPEATING THINGS
Chapter 1: The While Loop

DOING SOMETHING OVER AND OVER

Until now, each line of code ran once. If you wanted to print numbers 1 to 5, you wrote:

```
console.log(1)
console.log(2)
console.log(3)
console.log(4)
console.log(5)
```

What if you need to print 1 to 1000? Writing 1000 lines is not practical.

Loops solve this. A loop runs the same code multiple times.

The while loop is the simplest. It runs a block of code while a condition is true:

```
let count = 1

while (count <= 5) {
 console.log(count)
 count = count + 1
}
```

Output:
1
2
3
4
5

Five numbers printed with three lines of loop code.

HOW WHILE WORKS

The while loop has this structure:

```
while (condition) {
 code to repeat
}
```

JavaScript checks the condition. If true, it runs the code inside. Then it goes back to the condition and checks again. This continues until the condition is false.

Step by step through our example:

```
let count = 1

while (count <= 5) {
 console.log(count)
 count = count + 1
}
```

1. count starts at 1
2. Is 1 <= 5? Yes. Print 1. count becomes 2.
3. Is 2 <= 5? Yes. Print 2. count becomes 3.
4. Is 3 <= 5? Yes. Print 3. count becomes 4.
5. Is 4 <= 5? Yes. Print 4. count becomes 5.
6. Is 5 <= 5? Yes. Print 5. count becomes 6.
7. Is 6 <= 5? No. Stop the loop.

The loop ran 5 times, then the condition became false and it stopped.

THE THREE PARTS

Every loop needs three parts:

1. Initialization: Set up before the loop
2. Condition: Check before each iteration
3. Update: Change something so the condition eventually becomes false

```
let count = 1  (initialization)

while (count <= 5) {  (condition)
 console.log(count)
 count = count + 1  (update)
}
```

Miss any part and you have problems:

No initialization:

```
while (count <= 5) {
 console.log(count)
 count = count + 1
}
```

Error. count is not defined.

No condition:

```
let count = 1

while () {
 console.log(count)
 count = count + 1
}
```

Syntax error. While requires a condition.

No update:

```
let count = 1
```

```
while (count <= 5) {
 console.log(count)
}
```

Infinite loop. count is always 1, always <= 5. This runs forever (or until your computer stops it).

INFINITE LOOPS

An infinite loop never stops. The condition never becomes false.

```
while (true) {
 console.log("Forever")
}
```

This prints "Forever" endlessly. Your program freezes.

Common causes:

Forgetting to update the counter:

```
let i = 0
while (i < 10) {
 console.log(i)
}
```

i never changes. Always 0, always < 10.

Updating in the wrong direction:

```
let i = 0
while (i < 10) {
 console.log(i)
 i = i - 1
}
```

i goes 0, -1, -2, -3... further from 10, not toward it.

Condition that can never be false:

```
let x = 5
while (x > 0 || x < 10) {
 console.log(x)
 x = x + 1
}
```

Every number is either > 0 or < 10 (or both). This never stops.

If your program freezes, you probably have an infinite loop. In most environments, press Ctrl + C to stop it.

COUNTING PATTERNS

Count up:

```
let i = 1
while (i <= 10) {
 console.log(i)
 i = i + 1
}
```

Prints 1 through 10.

Count down:

```
let i = 10
while (i >= 1) {
 console.log(i)
 i = i - 1
}
```

Prints 10 through 1.

Count by twos:

```
let i = 2
while (i <= 20) {
 console.log(i)
 i = i + 2
}
```

Prints 2, 4, 6, 8, 10, 12, 14, 16, 18, 20.

Count by fives:

```
let i = 0
while (i <= 100) {
 console.log(i)
 i = i + 5
}
```

Prints 0, 5, 10, 15... up to 100.

PRACTICAL EXAMPLES

Calculate sum of 1 to 100:

```
let sum = 0
let i = 1

while (i <= 100) {
 sum = sum + i
 i = i + 1
}

console.log(`Sum of 1 to 100: ${sum}`)
```

Output: Sum of 1 to 100: 5050

Find factorial (5! = 5 * 4 * 3 * 2 * 1):

```
let n = 5
let factorial = 1
let i = 1

while (i <= n) {
 factorial = factorial * i
 i = i + 1
}

console.log(`${n}! = ${factorial}`)
```

Output: 5! = 120

Count digits in a number:

```
let number = 12345
let digitCount = 0
let temp = number

while (temp > 0) {
 digitCount = digitCount + 1
 temp = Math.floor(temp / 10)
}

console.log(`${number} has ${digitCount} digits`)
```

Output: 12345 has 5 digits

Repeat a character:

```
let char = "*"
let times = 5
let result = ""
let i = 0

while (i < times) {
 result = result + char
 i = i + 1
}
```

```
console.log(result)
```

Output: *****

WHILE WITH CONDITIONS OTHER THAN COUNTING

While is not just for counting. Any condition works:

```
let password = ""
let attempts = 0

while (password !== "secret" && attempts < 3) {
 attempts = attempts + 1
 password = "wrong"
 console.log(`Attempt ${attempts}: Incorrect password`)
}

if (password === "secret") {
 console.log("Access granted")
} else {
 console.log("Account locked")
}
```

Simulate finding a value:

```
let target = 7
let guess = 1

while (guess !== target) {
 console.log(`${guess} is not the target`)
 guess = guess + 1
}

console.log(`Found it: ${guess}`)
```

PRACTICE

Write a while loop that prints the even numbers from 2 to 20.

Write a while loop that calculates the sum of all odd numbers from 1 to 99.

Write a while loop that prints a countdown from 10 to 1, then prints "Blast off!"

Write a while loop that doubles a number until it exceeds 1000, starting from 1.

WHAT YOU LEARNED

You learned that while loops repeat code while a condition is true.
You learned that loops need initialization, condition, and update.
You learned that forgetting the update causes infinite loops.
You learned to count up, down, and by different increments.
You learned to use while for calculations like sums and factorials.

The next chapter teaches you a cleaner syntax for counting loops.

BOOK 3: REPEATING THINGS

Chapter 2: The For Loop

A CLEANER COUNTING LOOP

The while loop for counting looks like this:

```
let i = 0
while (i < 5) {
 console.log(i)
```

```
 i = i + 1
}
```

The three parts (initialization, condition, update) are spread across multiple lines. The for loop puts them all in one place:

```
for (let i = 0; i < 5; i = i + 1) {
 console.log(i)
}
```

Output:
0
1
2
3
4

Same result, but the loop mechanics are all visible in one line.

FOR LOOP STRUCTURE

```
for (initialization; condition; update) {
 code to repeat
}
```

The three parts are separated by semicolons:

1. Initialization: Runs once before the loop starts
2. Condition: Checked before each iteration
3. Update: Runs after each iteration

```
for (let i = 1; i <= 5; i = i + 1) {
 console.log(i)
}
```

Step by step:

1. let i = 1 runs once
2. Is 1 <= 5? Yes. Print 1. i becomes 2.
3. Is 2 <= 5? Yes. Print 2. i becomes 3.
4. Is 3 <= 5? Yes. Print 3. i becomes 4.
5. Is 4 <= 5? Yes. Print 4. i becomes 5.
6. Is 5 <= 5? Yes. Print 5. i becomes 6.
7. Is 6 <= 5? No. Stop.

SHORTHAND FOR UPDATE

Adding 1 to a variable is so common that there is shorthand:

i = i + 1 can be written as i++
i = i - 1 can be written as i--

```
for (let i = 1; i <= 5; i++) {
 console.log(i)
}
```

This is the most common for loop you will see. The ++ is called the increment operator. The -- is the decrement operator.

Other shorthand:

i += 2 means i = i + 2
i -= 3 means i = i - 3
i *= 2 means i = i * 2
i /= 2 means i = i / 2

```
for (let i = 0; i <= 20; i += 5) {
 console.log(i)
}
```

Output: 0, 5, 10, 15, 20

COMMON FOR LOOP PATTERNS

Count from 0 to n-1 (very common in programming):

```
for (let i = 0; i < 10; i++) {
 console.log(i)
}
```

Prints 0 through 9 (10 numbers total).

Count from 1 to n:

```
for (let i = 1; i <= 10; i++) {
 console.log(i)
}
```

Prints 1 through 10.

Count backwards:

```
for (let i = 10; i >= 1; i--) {
 console.log(i)
}
```

Prints 10 through 1.

Count by twos:

```
for (let i = 0; i <= 20; i += 2) {
 console.log(i)
}
```

Prints 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20.

Run exactly n times:

```
let n = 5

for (let i = 0; i < n; i++) {
```

```
 console.log(`Iteration ${i + 1}`)
}
```

Runs 5 times, printing "Iteration 1" through "Iteration 5".

USING THE LOOP VARIABLE

The variable i is available inside the loop. Use it for calculations:

```
for (let i = 1; i <= 5; i++) {
 let square = i * i
 console.log(`${i} squared is ${square}`)
}
```

Output:
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25

Build a pattern:

```
for (let i = 1; i <= 5; i++) {
 let stars = ""
 for (let j = 0; j < i; j++) {
 stars = stars + "*"
 }
 console.log(stars)
}
```

Output:
*
**
***
****
*****

This has a loop inside a loop. We will learn more about nested loops later.

PRACTICAL EXAMPLES

Sum of numbers:

```
let sum = 0

for (let i = 1; i <= 100; i++) {
 sum = sum + i
}

console.log(`Sum: ${sum}`)
```

Output: Sum: 5050

Factorial:

```
let n = 5
let factorial = 1

for (let i = 1; i <= n; i++) {
 factorial = factorial * i
}

console.log(`${n}! = ${factorial}`)
```

Output: 5! = 120

Powers of 2:

```
for (let i = 0; i <= 10; i++) {
 let power = Math.pow(2, i)
 console.log(`2^${i} = ${power}`)
}
```

Output:
2^0 = 1
2^1 = 2
2^2 = 4
...
2^10 = 1024

Multiplication table:

```
let number = 7

for (let i = 1; i <= 10; i++) {
 console.log(`${number} x ${i} = ${number * i}`)
}
```

Fizz Buzz (classic programming exercise):

```
for (let i = 1; i <= 20; i++) {
 if (i % 15 === 0) {
 console.log("FizzBuzz")
 } else if (i % 3 === 0) {
 console.log("Fizz")
 } else if (i % 5 === 0) {
 console.log("Buzz")
 } else {
 console.log(i)
 }
}
```

SCOPE OF THE LOOP VARIABLE

When you declare i with let inside the for loop, it only exists inside that loop:

```
for (let i = 0; i < 3; i++) {
 console.log(i)
```

```
}

console.log(i)
```

Error. i does not exist outside the loop.

If you need the final value after the loop:

```
let i
for (i = 0; i < 3; i++) {
 console.log(i)
}

console.log(`Loop ended at i = ${i}`)
```

Now i exists outside and has the value 3.

EMPTY PARTS

You can leave parts of the for loop empty:

```
let i = 0
for (; i < 5; i++) {
 console.log(i)
}
```

Initialization done before the loop.

```
let i = 0
for (; i < 5;) {
 console.log(i)
 i++
}
```

Update done inside the loop. This is basically a while loop.

```
for (;;) {
```

```
 console.log("Infinite loop")
}
```

All parts empty. No condition means always true. This is an infinite loop.

Usually you should include all three parts. Empty parts make code harder to read.

FOR VS WHILE

Use for when:
- You know how many times to loop
- You are counting
- You are iterating through a range

Use while when:
- You do not know how many iterations
- You are waiting for a condition to change
- The loop might not run at all

for is usually preferred for counting because it keeps initialization, condition, and update together.

PRACTICE

Write a for loop that prints the numbers 1 to 50.

Write a for loop that prints all multiples of 3 from 3 to 30.

Write a for loop that calculates 10 factorial (10!).

Write a for loop that prints the first 10 powers of 3 (3^1 through 3^10).

Write a for loop that prints a countdown from 20 to 0 by twos.

WHAT YOU LEARNED

You learned that for loops put initialization, condition, and update together.
You learned the structure: for (init; condition; update) { code }
You learned the ++ and -- shortcuts.
You learned += and -= shortcuts.
You learned common patterns like counting 0 to n-1.
You learned that the loop variable exists only inside the loop when declared with let.

The next chapter teaches you how to process collections of items.

BOOK 3: REPEATING THINGS

Chapter 3: Looping Through Arrays

ARRAYS HOLD MULTIPLE VALUES

Before we loop through arrays, you need to know what they are. An array is a list of values:

```
let fruits = ["apple", "banana", "orange"]
```

This array has three elements. Each element has a position (index) starting at 0:

Index 0: "apple"
Index 1: "banana"
Index 2: "orange"

Access elements with square brackets:

```
console.log(fruits[0])
console.log(fruits[1])
console.log(fruits[2])
```

Output:
apple
banana
orange

Arrays have a length property:

```
console.log(fruits.length)
```

Output: 3

Book 5 covers arrays in depth. For now, this is enough to understand looping.

LOOPING WITH A FOR LOOP

To do something with every element in an array, use a for loop:

```
let fruits = ["apple", "banana", "orange"]

for (let i = 0; i < fruits.length; i++) {
 console.log(fruits[i])
}
```

Output:
apple
banana
orange

Breaking it down:

let i = 0: Start at the first index (0)

i < fruits.length: Loop while i is less than 3
i++: Move to the next index
fruits[i]: Access the element at position i

When i is 0, fruits[0] is "apple".
When i is 1, fruits[1] is "banana".
When i is 2, fruits[2] is "orange".
When i is 3, the condition i < 3 is false. Loop stops.

WHY i < length NOT i <= length

Arrays are zero-indexed. An array with 3 elements has indices 0, 1, 2.

let arr = ["a", "b", "c"]

arr.length is 3
Valid indices are 0, 1, 2
arr[3] is undefined (does not exist)

Use < to stop before hitting an invalid index:

```
for (let i = 0; i < arr.length; i++) {
 console.log(arr[i])
}
```

If you use <=, you get undefined:

```
for (let i = 0; i <= arr.length; i++) {
 console.log(arr[i])
}
```

Output:
a
b
c
undefined

The last iteration accesses arr[3], which does not exist.

LOOPING BACKWARDS

Start at the last index and go down to 0:

```
let numbers = [10, 20, 30, 40, 50]

for (let i = numbers.length - 1; i >= 0; i--) {
 console.log(numbers[i])
}
```

Output:
50
40
30
20
10

numbers.length is 5. The last valid index is 4 (length - 1). We start there and go down to 0.

PRACTICAL EXAMPLES

Sum all numbers:

```
let prices = [10, 25, 5, 30, 15]
let total = 0

for (let i = 0; i < prices.length; i++) {
 total = total + prices[i]
}
```

```
console.log(`Total: $${total}`)
```

Output: Total: $85

Find the maximum:

```
let scores = [85, 92, 78, 95, 88]
let highest = scores[0]

for (let i = 1; i < scores.length; i++) {
 if (scores[i] > highest) {
 highest = scores[i]
 }
}

console.log(`Highest score: ${highest}`)
```

Output: Highest score: 95

Count specific items:

```
let responses = ["yes", "no", "yes", "yes", "no", "yes"]
let yesCount = 0

for (let i = 0; i < responses.length; i++) {
 if (responses[i] === "yes") {
 yesCount++
 }
}

console.log(`Yes votes: ${yesCount}`)
```

Output: Yes votes: 4

Find an item:

```
let names = ["Alice", "Bob", "Charlie", "David"]
let searchFor = "Charlie"
let foundIndex = -1
```

```
for (let i = 0; i < names.length; i++) {
 if (names[i] === searchFor) {
 foundIndex = i
 }
}

if (foundIndex !== -1) {
 console.log(`Found ${searchFor} at index ${foundIndex}`)
} else {
 console.log(`${searchFor} not found`)
}
```

Output: Found Charlie at index 2

THE FOR...OF LOOP

JavaScript has an easier way to loop through arrays when you do not need the index:

```
let fruits = ["apple", "banana", "orange"]

for (let fruit of fruits) {
 console.log(fruit)
}
```

Output:
apple
banana
orange

The for...of loop gives you each element directly. No index variable, no length, no brackets.

The syntax:

```
for (let element of array) {
 use element
```

}

You choose the name for element. It represents each item as you loop.

```
let numbers = [1, 2, 3, 4, 5]
let sum = 0

for (let num of numbers) {
 sum = sum + num
}

console.log(`Sum: ${sum}`)
```

Output: Sum: 15

Use for...of when:
- You just need the values
- You do not need to know the index
- You are not modifying the array

Use the traditional for loop when:
- You need the index for something
- You need to modify elements
- You want to skip certain indices

LOOPING THROUGH STRINGS

Strings work like arrays of characters:

```
let word = "Hello"

for (let i = 0; i < word.length; i++) {
 console.log(word[i])
}
```

Output:

```
H
e
l
l
o
```

For...of also works with strings:

```
let word = "Hello"

for (let char of word) {
 console.log(char)
}
```

Same output.

Reverse a string:

```
let word = "Hello"
let reversed = ""

for (let i = word.length - 1; i >= 0; i--) {
 reversed = reversed + word[i]
}

console.log(reversed)
```

Output: olleH

Count vowels:

```
let text = "JavaScript is awesome"
let vowelCount = 0
let vowels = "aeiouAEIOU"

for (let char of text) {
 if (vowels.includes(char)) {
 vowelCount++
 }
}
```

```
console.log(`Vowels: ${vowelCount}`)
```

Output: Vowels: 7

COMMON MISTAKES

Off by one errors:

```
for (let i = 1; i <= fruits.length; i++) {
 console.log(fruits[i])
}
```

This skips index 0 and accesses an invalid index. Use `i = 0` and `i < length`.

Using for...of when you need the index:

```
for (let fruit of fruits) {
 console.log(`Index ???: ${fruit}`)
}
```

You cannot get the index in for...of. Use the traditional for loop.

Modifying array during loop:

```
let numbers = [1, 2, 3, 4, 5]

for (let i = 0; i < numbers.length; i++) {
 if (numbers[i] === 3) {
 numbers.splice(i, 1)
 }
}
```

Removing elements while looping causes problems. The indices shift. Loop backwards or use filter instead.

PRACTICE

Loop through an array of names and print a greeting for each.

Loop through an array of prices and calculate the total.

Loop through a string and count how many times the letter "a" appears.

Loop through an array of numbers and create a new array with only the even numbers.

Loop through an array and find the smallest number.

WHAT YOU LEARNED

You learned that arrays hold multiple values with numbered indices.
You learned to loop through arrays with for (let i = 0; i < arr.length; i++).
You learned why we use < instead of <= with length.
You learned the for...of loop for simpler iteration.
You learned that strings can be looped like arrays.
You learned common operations: sum, max, min, count, search.

The next chapter teaches you how to control loops with break and continue.

BOOK 3: REPEATING THINGS

Chapter 4: Break and Continue

STOPPING A LOOP EARLY

Sometimes you want to exit a loop before it finishes naturally. The break statement does this:

```
for (let i = 1; i <= 10; i++) {
 console.log(i)
 if (i === 5) {
 break
 }
}

console.log("Loop ended")
```

Output:
1
2
3
4
5
Loop ended

The loop was set to run from 1 to 10. But when i reached 5, break stopped the loop immediately. Numbers 6 through 10 were never printed.

WHEN TO USE BREAK

Break is useful when you find what you are looking for:

```
let names = ["Alice", "Bob", "Charlie", "David", "Eve"]
let searchFor = "Charlie"
let found = false
let foundIndex = -1

for (let i = 0; i < names.length; i++) {
 if (names[i] === searchFor) {
```

```
 found = true
 foundIndex = i
 break
 }
}

if (found) {
 console.log(`Found ${searchFor} at index ${foundIndex}`)
}
```

Output: Found Charlie at index 2

Once we find Charlie, there is no need to check the rest of the array. Break exits immediately.

Without break, the loop would check David and Eve unnecessarily. For small arrays this does not matter. For arrays with millions of elements, it matters a lot.

BREAK WITH WHILE LOOPS

Break works with any loop:

```
let count = 0

while (true) {
 count++
 console.log(count)

 if (count >= 5) {
 break
 }
}

console.log("Done")
```

Output:

```
1
2
3
4
5
Done
```

The condition is always true, which would normally create an infinite loop. But break provides an exit when count reaches 5.

This pattern is sometimes clearer than putting the condition in the while:

```
let input = ""
let attempts = 0

while (true) {
 attempts++
 input = "wrong"

 if (input === "correct") {
 console.log("Success!")
 break
 }

 if (attempts >= 3) {
 console.log("Too many attempts")
 break
 }

 console.log(`Attempt ${attempts} failed`)
}
```

Multiple exit conditions are easy to see with break.

SKIPPING TO THE NEXT ITERATION

The continue statement skips the rest of the current iteration and moves to the next one:

```
for (let i = 1; i <= 5; i++) {
 if (i === 3) {
 continue
 }
 console.log(i)
}
```

Output:
1
2
4
5

When i was 3, continue skipped the console.log and went straight to the next iteration (i = 4).

WHEN TO USE CONTINUE

Continue is useful when you want to skip certain items:

```
let numbers = [1, -2, 3, -4, 5, -6, 7]

console.log("Positive numbers only:")

for (let num of numbers) {
 if (num < 0) {
 continue
 }
 console.log(num)
}
```

Output:
Positive numbers only:
1

3
5
7

Negative numbers are skipped. Only positive numbers are printed.

Process only valid items:

```
let data = ["apple", "", "banana", null, "cherry", undefined, "date"]

for (let item of data) {
 if (!item) {
 continue
 }
 console.log(`Processing: ${item}`)
}
```

Output:
Processing: apple
Processing: banana
Processing: cherry
Processing: date

Empty strings, null, and undefined are skipped.

CONTINUE VS IF/ELSE

These two approaches do the same thing:

Using continue:

```
for (let i = 1; i <= 10; i++) {
 if (i % 2 !== 0) {
 continue
 }
 console.log(i)
}
```

Using if:

```
for (let i = 1; i <= 10; i++) {
 if (i % 2 === 0) {
 console.log(i)
 }
}
```

Both print even numbers 2, 4, 6, 8, 10.

Continue is useful when you have many lines of code after the check. It avoids deep nesting:

Without continue (deeply nested):

```
for (let item of items) {
 if (item.isValid) {
 if (item.isActive) {
 if (item.hasPermission) {
 processItem(item)
 logItem(item)
 updateItem(item)
 }
 }
 }
}
```

With continue (flat structure):

```
for (let item of items) {
 if (!item.isValid) continue
 if (!item.isActive) continue
 if (!item.hasPermission) continue

 processItem(item)
 logItem(item)
 updateItem(item)
}
```

The second version is easier to read. Each continue acts as a guard clause.

BREAK VS CONTINUE

Break: Exit the entire loop. Done. Finished. No more iterations.

Continue: Skip this iteration. Go to the next one. Loop keeps running.

```
for (let i = 1; i <= 5; i++) {
 if (i === 3) break
 console.log(i)
}
```

Output: 1, 2 (loop stops at 3)

```
for (let i = 1; i <= 5; i++) {
 if (i === 3) continue
 console.log(i)
}
```

Output: 1, 2, 4, 5 (skips 3, continues to 4 and 5)

BREAK AND CONTINUE WITH NESTED LOOPS

Break and continue only affect the innermost loop:

```
for (let i = 1; i <= 3; i++) {
 console.log(`Outer: ${i}`)

 for (let j = 1; j <= 3; j++) {
 if (j === 2) break
 console.log(`  Inner: ${j}`)
```

```
 }
}
```

Output:

```
Outer: 1
 Inner: 1
Outer: 2
 Inner: 1
Outer: 3
 Inner: 1
```

Break stops the inner loop when j is 2. But the outer loop continues.

To break out of multiple loops, use a labeled statement:

```
outerLoop: for (let i = 1; i <= 3; i++) {
 for (let j = 1; j <= 3; j++) {
 console.log(`i=${i}, j=${j}`)
 if (i === 2 && j === 2) {
 break outerLoop
 }
 }
}

console.log("Done")
```

Output:

```
i=1, j=1
i=1, j=2
i=1, j=3
i=2, j=1
i=2, j=2
Done
```

The label outerLoop names the outer for loop. break outerLoop exits that loop entirely.

PRACTICAL EXAMPLES

Find first matching item:

```
let products = [
 { name: "Apple", price: 1 },
 { name: "Banana", price: 2 },
 { name: "Cherry", price: 3 },
 { name: "Date", price: 4 }
]

let affordable = null

for (let product of products) {
 if (product.price <= 2) {
 affordable = product
 break
 }
}

console.log(`First affordable: ${affordable.name}`)
```

Validate input:

```
let inputs = ["valid", "valid", "invalid", "valid"]
let allValid = true

for (let input of inputs) {
 if (input === "invalid") {
 allValid = false
 break
 }
}

console.log(`All valid: ${allValid}`)
```

Skip weekends:

```
let days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
```

```
console.log("Work days:")
for (let day of days) {
 if (day === "Sat" || day === "Sun") {
 continue
 }
 console.log(day)
}
```

Process until error:

```
let operations = ["read", "process", "error", "save"]

for (let op of operations) {
 if (op === "error") {
 console.log("Error encountered, stopping")
 break
 }
 console.log(`Executing: ${op}`)
}
```

PRACTICE

Write a loop that finds the first negative number in an array and stops.

Write a loop that prints only numbers greater than 10, skipping others.

Write a loop that stops when the sum of numbers exceeds 100.

Write a loop that skips any string shorter than 3 characters.

WHAT YOU LEARNED

You learned that break exits a loop immediately.
You learned that continue skips to the next iteration.
You learned to use break when you find what you need.
You learned to use continue to skip unwanted items.
You learned that break and continue affect only the innermost loop.
You learned about labeled statements for breaking outer loops.

The next chapter teaches you about nested loops.

BOOK 3: REPEATING THINGS

Chapter 5: Nested Loops

A LOOP INSIDE A LOOP

You can put a loop inside another loop. The inner loop runs completely for each iteration of the outer loop:

```
for (let i = 1; i <= 3; i++) {
 console.log(`Outer: ${i}`)

 for (let j = 1; j <= 2; j++) {
 console.log(`  Inner: ${j}`)
 }
}
```

Output:

```
Outer: 1
 Inner: 1
 Inner: 2
Outer: 2
 Inner: 1
 Inner: 2
Outer: 3
 Inner: 1
 Inner: 2
```

For each iteration of the outer loop (i = 1, 2, 3), the inner loop

runs completely (j = 1, 2). The inner loop runs 2 times for each of 3 outer iterations. Total: 6 inner iterations.

HOW NESTED LOOPS EXECUTE

Think of it like clock hands:

The outer loop is the hour hand. It moves slowly.
The inner loop is the minute hand. It goes around completely before the hour hand moves.

```
for (let hour = 1; hour <= 3; hour++) {
 for (let minute = 0; minute < 60; minute += 15) {
 console.log(`${hour}:${minute.toString().padStart(2, "0")}`)
 }
}
```

Output:
1:00
1:15
1:30
1:45
2:00
2:15
2:30
2:45
3:00
3:15
3:30
3:45

For each hour, all the minutes print. Then the hour advances.

MULTIPLICATION TABLE

A classic use of nested loops:

```
for (let row = 1; row <= 5; row++) {
 let line = ""

 for (let col = 1; col <= 5; col++) {
 let product = row * col
 line = line + product + "\t"
 }

 console.log(line)
}
```

Output:

```
1        2        3        4        5
2        4        6        8        10
3        6        9        12       15
4        8        12       16       20
5        10       15       20       25
```

The outer loop handles rows. The inner loop builds each row by calculating products.

PATTERNS WITH NESTED LOOPS

Right triangle:

```
for (let row = 1; row <= 5; row++) {
 let stars = ""

 for (let col = 1; col <= row; col++) {
 stars = stars + "*"
 }

 console.log(stars)
}
```

Output:

```
*
**
***
****
*****
```

The inner loop runs row times. When row is 1, one star. When row is 5, five stars.

Inverted triangle:

```
for (let row = 5; row >= 1; row--) {
 let stars = ""

 for (let col = 1; col <= row; col++) {
 stars = stars + "*"
 }

 console.log(stars)
}
```

Output:

```
*****
****
***
**
*
```

Square:

```
let size = 4

for (let row = 1; row <= size; row++) {
 let line = ""

 for (let col = 1; col <= size; col++) {
 line = line + "* "
 }
```

```
 console.log(line)
}
```

Output:

```
* * * *
* * * *
* * * *
* * * *
```

Number pyramid:

```
for (let row = 1; row <= 5; row++) {
 let line = ""

 for (let col = 1; col <= row; col++) {
 line = line + col
 }

 console.log(line)
}
```

Output:

```
1
12
123
1234
12345
```

NESTED LOOPS WITH ARRAYS

Process a 2D array (array of arrays):

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
```

```
]

for (let row = 0; row < matrix.length; row++) {
 for (let col = 0; col < matrix[row].length; col++) {
 console.log(`matrix[${row}][${col}] = ${matrix[row][col]}`)
 }
}
```

Output:

```
matrix[0][0] = 1
matrix[0][1] = 2
matrix[0][2] = 3
matrix[1][0] = 4
matrix[1][1] = 5
matrix[1][2] = 6
matrix[2][0] = 7
matrix[2][1] = 8
matrix[2][2] = 9
```

Sum all elements in a 2D array:

```
let grid = [
 [10, 20, 30],
 [40, 50, 60]
]

let total = 0

for (let row of grid) {
 for (let cell of row) {
 total = total + cell
 }
}

console.log(`Total: ${total}`)
```

Output: Total: 210

COMPARING EVERY PAIR

Find all pairs in an array:

```
let names = ["Alice", "Bob", "Charlie"]

for (let i = 0; i < names.length; i++) {
 for (let j = 0; j < names.length; j++) {
 if (i !== j) {
 console.log(`${names[i]} meets ${names[j]}`)
 }
 }
}
```

Output:
Alice meets Bob
Alice meets Charlie
Bob meets Alice
Bob meets Charlie
Charlie meets Alice
Charlie meets Bob

Find unique pairs (without repeats):

```
let names = ["Alice", "Bob", "Charlie"]

for (let i = 0; i < names.length; i++) {
 for (let j = i + 1; j < names.length; j++) {
 console.log(`${names[i]} and ${names[j]}`)
 }
}
```

Output:
Alice and Bob
Alice and Charlie
Bob and Charlie

Starting j at i + 1 avoids duplicates. We only compare each pair once.

PRACTICAL EXAMPLES

Find common elements:

```
let arr1 = [1, 2, 3, 4, 5]
let arr2 = [3, 4, 5, 6, 7]
let common = []

for (let a of arr1) {
 for (let b of arr2) {
 if (a === b) {
 common.push(a)
 }
 }
}

console.log(`Common elements: ${common}`)
```

Output: Common elements: 3,4,5

Check for duplicates:

```
let items = ["apple", "banana", "apple", "cherry"]
let hasDuplicates = false

for (let i = 0; i < items.length; i++) {
 for (let j = i + 1; j < items.length; j++) {
 if (items[i] === items[j]) {
 console.log(`Duplicate found: ${items[i]}`)
 hasDuplicates = true
 }
 }
}

console.log(`Has duplicates: ${hasDuplicates}`)
```

Simple search in 2D:

```
let maze = [
 [".", ".", "#"],
 [".", "X", "."],
 ["#", ".", "."]
]

let targetRow = -1
let targetCol = -1

for (let row = 0; row < maze.length; row++) {
 for (let col = 0; col < maze[row].length; col++) {
 if (maze[row][col] === "X") {
 targetRow = row
 targetCol = col
 }
 }
}

console.log(`X found at row ${targetRow}, col ${targetCol}`)
```

Output: X found at row 1, col 1

PERFORMANCE CONSIDERATION

Nested loops can be slow. If the outer loop runs n times and the inner loop runs m times, the total is n times m iterations.

Two nested loops over an array of 1000 items:

```
for (let i = 0; i < 1000; i++) {
 for (let j = 0; j < 1000; j++) {
 doSomething()
 }
}
```

This runs 1,000,000 times. Add a third loop and you get 1,000,000,000 iterations.

For small data, this is fine. For large data, consider more efficient approaches. We will cover this in later books.

THREE LEVELS DEEP

You can nest more than two loops:

```
for (let x = 1; x <= 2; x++) {
 for (let y = 1; y <= 2; y++) {
 for (let z = 1; z <= 2; z++) {
 console.log(`(${x}, ${y}, ${z})`)
 }
 }
}
```

Output:
(1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 2, 2)
(2, 1, 1)
(2, 1, 2)
(2, 2, 1)
(2, 2, 2)

8 combinations (2 * 2 * 2).

But three or more levels of nesting becomes hard to read. If you find yourself going deeper, consider breaking the code into functions.

PRACTICE

Write nested loops that print a 5x5 grid of X characters.

Write nested loops that create a checkerboard pattern of # and spaces.

Write nested loops that find all pairs of numbers in an array that add up to 10.

Write nested loops that transpose a 2D array (swap rows and columns).

WHAT YOU LEARNED

You learned that loops can be placed inside other loops.
You learned that the inner loop runs completely for each outer iteration.
You learned to create patterns like triangles and squares.
You learned to process 2D arrays with nested loops.
You learned to compare pairs of elements.
You learned that nested loops multiply the iteration count.

The next chapter teaches you an alternative looping construct.

BOOK 3: REPEATING THINGS

Chapter 6: Do While Loops

RUN AT LEAST ONCE

The while loop checks the condition before running:

```
let count = 10
```

```
while (count < 5) {
 console.log(count)
 count++
}

console.log("Done")
```

Output: Done

The condition count < 5 is false immediately (10 is not less than 5). The loop body never runs.

Sometimes you want the code to run at least once, then check the condition. The do...while loop does this:

```
let count = 10

do {
 console.log(count)
 count++
} while (count < 5)

console.log("Done")
```

Output:
10
Done

The body ran once. Then the condition was checked. 11 is not less than 5, so the loop stopped.

DO WHILE STRUCTURE

```
do {
 code to run
} while (condition)
```

Key difference from while:

while: Check first, then maybe run.
do...while: Run first, then check for more runs.

The do...while always executes at least once, even if the condition starts false.

WHEN DO WHILE IS USEFUL

User input validation:

```
let userInput = ""
let attempts = 0

do {
 attempts++
 userInput = "invalid"
 console.log(`Attempt ${attempts}: ${userInput}`)
} while (userInput !== "valid" && attempts < 3)
```

Output:
Attempt 1: invalid
Attempt 2: invalid
Attempt 3: invalid

You must ask for input at least once before you can check if it is valid.

Menu systems:

```
let choice = 0

do {
 console.log("1. Start Game")
 console.log("2. Options")
 console.log("3. Exit")
```

```
 choice = 3
 console.log(`Selected: ${choice}`)
} while (choice !== 3)

console.log("Goodbye!")
```

The menu displays at least once before checking if the user wants to exit.

Game loops:

```
let gameOver = false
let turn = 0

do {
 turn++
 console.log(`Turn ${turn}`)

 if (turn >= 5) {
 gameOver = true
 }
} while (!gameOver)

console.log("Game Over")
```

The game runs at least one turn.

COMPARING WHILE AND DO WHILE

When condition starts true, they behave the same:

```
let i = 0

while (i < 3) {
 console.log(i)
 i++
}
```

Output: 0, 1, 2

```
let j = 0

do {
 console.log(j)
 j++
} while (j < 3)
```

Output: 0, 1, 2

When condition starts false:

```
let i = 5

while (i < 3) {
 console.log("while: " + i)
 i++
}
```

No output. Condition is false from the start.

```
let j = 5

do {
 console.log("do while: " + j)
 j++
} while (j < 3)
```

Output: do while: 5

The body ran once before checking.

PRACTICAL EXAMPLES

Generate random number until target:

```
let target = 7
let guess

do {
 guess = Math.floor(Math.random() * 10) + 1
 console.log(`Generated: ${guess}`)
} while (guess !== target)

console.log(`Hit target: ${target}`)
```

At least one number is generated, then we check if it matches.

Reading until end marker:

```
let data = ["item1", "item2", "item3", "END", "item4"]
let index = 0
let current

do {
 current = data[index]
 console.log(`Processing: ${current}`)
 index++
} while (current !== "END" && index < data.length)
```

Output:

```
Processing: item1
Processing: item2
Processing: item3
Processing: END
```

Countdown with message:

```
let fuel = 3

do {
 console.log(`Fuel: ${fuel}`)
 fuel--
} while (fuel > 0)
```

```
console.log("Empty!")
```

Output:
Fuel: 3
Fuel: 2
Fuel: 1
Empty!

Password verification with retries:

```
let password = ""
let correctPassword = "secret"
let attempts = 0
let maxAttempts = 3
let accessGranted = false

do {
 attempts++
 password = attempts === 3 ? "secret" : "wrong"
 console.log(`Attempt ${attempts}: ${password}`)

 if (password === correctPassword) {
 accessGranted = true
 }
} while (!accessGranted && attempts < maxAttempts)

if (accessGranted) {
 console.log("Access granted!")
} else {
 console.log("Account locked")
}
```

DO WHILE VS OTHER LOOPS

Use do...while when:
- The code must run at least once
- You need to get input before validating

- You are implementing a "repeat until" pattern

Use while when:
- The condition might be false initially
- Zero iterations is a valid outcome
- You are implementing a "while true" pattern

Use for when:
- You know how many iterations
- You are counting or iterating through indices

COMMON MISTAKE

Forgetting the semicolon:

```
do {
 console.log("Hello")
} while (condition)
```

The while line needs a semicolon at the end. Unlike regular while loops, do...while ends with a semicolon after the condition.

```
do {
 console.log("Hello")
} while (false);
```

The semicolon tells JavaScript the statement is complete.

CONVERTING BETWEEN WHILE AND DO WHILE

Any do...while can be written as while with duplication:

do...while version:

```
do {
 processItem()
} while (hasMoreItems())
```

while version (duplicated first call):

```
processItem()
while (hasMoreItems()) {
 processItem()
}
```

The while version calls processItem twice in the code. The do...while version is cleaner.

Any while can be written as do...while with extra check:

while version:

```
while (condition) {
 doSomething()
}
```

do...while version (with guard):

```
if (condition) {
 do {
 doSomething()
 } while (condition)
}
```

The do...while version is more complex. Use while when the condition might start false.

PRACTICE

Write a do...while loop that counts from 1 to 5.

Write a do...while loop that divides a number by 2 until it becomes less than 1.

Write a do...while loop that simulates rolling a die until you get a 6.

Write a do...while loop that builds a string by adding letters until the string is 10 characters long.

WHAT YOU LEARNED

You learned that do...while runs the body first, then checks the condition.
You learned that do...while always executes at least once.
You learned common use cases: input validation, menus, game loops.
You learned to choose do...while when one iteration is guaranteed.
You learned that do...while ends with a semicolon after the condition.

The next chapter teaches you loop-related methods for arrays.

BOOK 3: REPEATING THINGS

Chapter 7: Loop Methods

ARRAYS HAVE BUILT-IN LOOPS

JavaScript arrays come with methods that loop for you. These are often cleaner than manual loops.

Instead of:

```
let numbers = [1, 2, 3, 4, 5]

for (let i = 0; i < numbers.length; i++) {
```

```
 console.log(numbers[i])
}
```

You can write:

```
let numbers = [1, 2, 3, 4, 5]

numbers.forEach(function(num) {
 console.log(num)
})
```

Same output, different style.

FOREACH

The forEach method runs a function for each element:

```
let fruits = ["apple", "banana", "cherry"]

fruits.forEach(function(fruit) {
 console.log(fruit)
})
```

Output:
apple
banana
cherry

The function inside is called for each element. The parameter fruit receives each element in turn.

You can also get the index:

```
fruits.forEach(function(fruit, index) {
 console.log(`${index}: ${fruit}`)
})
```

Output:
0: apple
1: banana
2: cherry

And the original array:

```
fruits.forEach(function(fruit, index, array) {
 console.log(`Item ${index + 1} of ${array.length}: ${fruit}`)
})
```

Output:
Item 1 of 3: apple
Item 2 of 3: banana
Item 3 of 3: cherry

ARROW FUNCTIONS

Modern JavaScript has a shorter way to write functions:

Traditional function:

```
fruits.forEach(function(fruit) {
 console.log(fruit)
})
```

Arrow function:

```
fruits.forEach((fruit) = > {
 console.log(fruit)
})
```

Even shorter for single expressions:

```
fruits.forEach(fruit = > console.log(fruit))
```

The arrow function removes the function keyword and uses = >

instead. For single parameters, parentheses are optional. For single-line bodies, braces and return are optional.

All three do the same thing.

MAP: TRANSFORM EACH ELEMENT

The map method creates a new array by transforming each element:

```
let numbers = [1, 2, 3, 4, 5]

let doubled = numbers.map(function(num) {
 return num * 2
})

console.log(doubled)
```

Output: [2, 4, 6, 8, 10]

Map runs the function for each element and collects the returned values into a new array.

With arrow function:

```
let doubled = numbers.map(num => num * 2)
```

Much cleaner.

More examples:

```
let names = ["alice", "bob", "charlie"]

let upperNames = names.map(name => name.toUpperCase())

console.log(upperNames)
```

Output: ["ALICE", "BOB", "CHARLIE"]

```
let prices = [10, 20, 30]

let withTax = prices.map(price => price * 1.1)

console.log(withTax)
```

Output: [11, 22, 33]

Map does not change the original array. It creates a new one:

```
let original = [1, 2, 3]
let modified = original.map(x => x * 10)

console.log(original)
console.log(modified)
```

Output:
[1, 2, 3]
[10, 20, 30]

FILTER: KEEP MATCHING ELEMENTS

The filter method creates a new array with only elements that pass a test:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let evens = numbers.filter(function(num) {
 return num % 2 === 0
})

console.log(evens)
```

Output: [2, 4, 6, 8, 10]

The function returns true or false for each element. Elements that

return true are kept.

With arrow function:

```
let evens = numbers.filter(num => num % 2 === 0)
```

More examples:

```
let words = ["apple", "bee", "cherry", "date", "elderberry"]

let longWords = words.filter(word => word.length > 5)

console.log(longWords)
```

Output: ["cherry", "elderberry"]

```
let temperatures = [72, 85, 68, 91, 76, 88]

let hotDays = temperatures.filter(temp => temp >= 85)

console.log(hotDays)
```

Output: [85, 91, 88]

Filter does not change the original array:

```
let all = [1, 2, 3, 4, 5]
let big = all.filter(n => n > 3)

console.log(all)
console.log(big)
```

Output:
[1, 2, 3, 4, 5]
[4, 5]

CHAINING METHODS

You can chain map and filter together:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

let result = numbers
 .filter(n => n % 2 === 0)
 .map(n => n * n)

console.log(result)
```

Output: [4, 16, 36, 64, 100]

First filter keeps even numbers [2, 4, 6, 8, 10].
Then map squares them [4, 16, 36, 64, 100].

The order matters:

```
let result2 = numbers
 .map(n => n * n)
 .filter(n => n > 25)

console.log(result2)
```

Output: [36, 49, 64, 81, 100]

First squares all [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].
Then filters those greater than 25.

REDUCE: COMBINE ALL ELEMENTS

The reduce method combines all elements into a single value:

```
let numbers = [1, 2, 3, 4, 5]

let sum = numbers.reduce(function(accumulator, current) {
 return accumulator + current
```

}, 0)

console.log(sum)

Output: 15

The function takes two parameters:
- accumulator: the running total
- current: the current element

The second argument (0) is the starting value.

Step by step:
- Start: accumulator = 0
- Add 1: accumulator = 0 + 1 = 1
- Add 2: accumulator = 1 + 2 = 3
- Add 3: accumulator = 3 + 3 = 6
- Add 4: accumulator = 6 + 4 = 10
- Add 5: accumulator = 10 + 5 = 15

With arrow function:

let sum = numbers.reduce((acc, cur) => acc + cur, 0)

More examples:

Product of all numbers:

let product = [2, 3, 4].reduce((acc, cur) => acc * cur, 1)

console.log(product)

Output: 24 (2 * 3 * 4)

Find maximum:

let max = [5, 2, 9, 1, 7].reduce((acc, cur) => cur > acc ? cur : acc)

console.log(max)

Output: 9

Count occurrences:

```
let letters = ["a", "b", "a", "c", "a", "b"]

let counts = letters.reduce((acc, letter) => {
 acc[letter] = (acc[letter] || 0) + 1
 return acc
}, {})

console.log(counts)
```

Output: {a: 3, b: 2, c: 1}

FIND: GET FIRST MATCH

The find method returns the first element that matches:

```
let numbers = [1, 5, 10, 15, 20]

let firstBig = numbers.find(n => n > 8)

console.log(firstBig)
```

Output: 10

Unlike filter (which returns all matches), find returns only the first one.

If nothing matches, find returns undefined:

```
let notFound = numbers.find(n => n > 100)

console.log(notFound)
```

Output: undefined

SOME AND EVERY

some checks if at least one element matches:

let numbers = [1, 2, 3, 4, 5]

let hasEven = numbers.some(n => n % 2 === 0)

console.log(hasEven)

Output: true

every checks if all elements match:

let allPositive = numbers.every(n => n > 0)

console.log(allPositive)

Output: true

let allBig = numbers.every(n => n > 3)

console.log(allBig)

Output: false

WHEN TO USE WHAT

forEach: When you want to do something with each element (side effects)
map: When you want to transform each element into something new

filter: When you want to keep only certain elements
reduce: When you want to combine elements into one value
find: When you want the first matching element
some: When you want to know if any element matches
every: When you want to know if all elements match

PRACTICE

Use forEach to print each item in a shopping list.

Use map to convert an array of Celsius temperatures to Fahrenheit.

Use filter to get only positive numbers from an array.

Use reduce to count the total length of all strings in an array.

Use find to get the first number greater than 100.

WHAT YOU LEARNED

You learned that forEach runs a function for each element.
You learned that map creates a new array by transforming elements.
You learned that filter creates a new array with matching elements.
You learned that reduce combines elements into a single value.
You learned arrow function syntax for cleaner code.
You learned to chain methods together.

The next chapter brings everything together with practice programs.
BOOK 3: REPEATING THINGS

Chapter 8: Practice Loop Programs

NUMBER ANALYZER

Analyze a collection of numbers:

```
let numbers = [23, 45, 12, 67, 34, 89, 56, 78, 90, 11]

console.log("=== NUMBER ANALYZER ===")
console.log(`Numbers: ${numbers.join(", ")}`)
console.log("")

let sum = 0
let min = numbers[0]
let max = numbers[0]
let evenCount = 0
let oddCount = 0

for (let num of numbers) {
 sum = sum + num

 if (num < min) {
 min = num
 }

 if (num > max) {
 max = num
 }

 if (num % 2 === 0) {
 evenCount++
 } else {
 oddCount++
 }
}

let average = sum / numbers.length
let range = max - min
```

```
console.log(`Count: ${numbers.length}`)
console.log(`Sum: ${sum}`)
console.log(`Average: ${average}`)
console.log(`Minimum: ${min}`)
console.log(`Maximum: ${max}`)
console.log(`Range: ${range}`)
console.log(`Even numbers: ${evenCount}`)
console.log(`Odd numbers: ${oddCount}`)

let above = numbers.filter(n => n > average)
let below = numbers.filter(n => n <= average)

console.log("")
console.log(`Above average: ${above.join(", ")}`)
console.log(`At or below average: ${below.join(", ")}`)
```

WORD PROCESSOR

Process and analyze text:

```
let text = "The quick brown fox jumps over the lazy dog"

console.log("=== WORD PROCESSOR ===")
console.log(`Text: "${text}"`)
console.log("")

let words = text.split(" ")
let charCount = 0
let wordLengths = []
let longestWord = ""
let shortestWord = words[0]

for (let word of words) {
 charCount = charCount + word.length
 wordLengths.push(word.length)

 if (word.length > longestWord.length) {
```

```
 longestWord = word
 }

 if (word.length < shortestWord.length) {
 shortestWord = word
 }
}

let avgLength = charCount / words.length

console.log(`Word count: ${words.length}`)
console.log(`Character count (no spaces): ${charCount}`)
console.log(`Average word length: ${avgLength.toFixed(2)}`)
console.log(`Longest word: "${longestWord}"
(${longestWord.length} chars)`)
console.log(`Shortest word: "${shortestWord}"
(${shortestWord.length} chars)`)
console.log("")

let vowelCount = 0
let consonantCount = 0
let vowels = "aeiouAEIOU"
let consonants =
"bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ"

for (let char of text) {
 if (vowels.includes(char)) {
 vowelCount++
 } else if (consonants.includes(char)) {
 consonantCount++
 }
}

console.log(`Vowels: ${vowelCount}`)
console.log(`Consonants: ${consonantCount}`)

let reversed = words.map(word =>
word.split("").reverse().join(""))

console.log("")
```

```
console.log(`Reversed words: ${reversed.join(" ")}`)
```

SHOPPING CART

Complete shopping cart with totals and discounts:

```
let cart = [
 { name: "Laptop", price: 999.99, quantity: 1 },
 { name: "Mouse", price: 29.99, quantity: 2 },
 { name: "Keyboard", price: 79.99, quantity: 1 },
 { name: "Monitor", price: 299.99, quantity: 2 },
 { name: "USB Cable", price: 9.99, quantity: 5 }
]

console.log("=== SHOPPING CART ===")
console.log("")

let subtotal = 0
let itemCount = 0

for (let item of cart) {
 let lineTotal = item.price * item.quantity
 subtotal = subtotal + lineTotal
 itemCount = itemCount + item.quantity

 console.log(`${item.name}`)
 console.log(`  $${item.price} x ${item.quantity} = $
${lineTotal.toFixed(2)}`)
}

console.log("")
console.log(`Subtotal: $${subtotal.toFixed(2)}`)
console.log(`Total items: ${itemCount}`)

let discount = 0
let discountReason = ""
```

```
if (subtotal >= 500) {
 discount = subtotal * 0.1
 discountReason = "10% off orders over $500"
} else if (subtotal >= 100) {
 discount = subtotal * 0.05
 discountReason = "5% off orders over $100"
}

if (discount > 0) {
 console.log(`Discount (${discountReason}): -$
${discount.toFixed(2)}`)
}

let tax = (subtotal - discount) * 0.08
let total = subtotal - discount + tax

console.log(`Tax (8%): $${tax.toFixed(2)}`)
console.log(`Total: $${total.toFixed(2)}`)

let expensive = cart.filter(item => item.price > 100)
let cheap = cart.filter(item => item.price <= 20)

console.log("")
console.log(`Premium items (over $100): ${expensive.map(i =>
i.name).join(", ")}`)
console.log(`Budget items (under $20): ${cheap.map(i =>
i.name).join(", ")}`)
```

PRIME NUMBER FINDER

Find prime numbers in a range:

```
let start = 1
let end = 100

console.log("=== PRIME NUMBER FINDER ===")
console.log(`Range: ${start} to ${end}`)
```

```
console.log("")

let primes = []

for (let num = start; num <= end; num++) {
 if (num < 2) {
 continue
 }

 let isPrime = true

 for (let divisor = 2; divisor <= Math.sqrt(num); divisor++) {
 if (num % divisor === 0) {
 isPrime = false
 break
 }
 }

 if (isPrime) {
 primes.push(num)
 }
}

console.log(`Found ${primes.length} prime numbers:`)
console.log(primes.join(", "))
console.log("")

let twinPrimes = []

for (let i = 0; i < primes.length - 1; i++) {
 if (primes[i + 1] - primes[i] === 2) {
 twinPrimes.push(`(${primes[i]}, ${primes[i + 1]})`)
 }
}

console.log(`Twin primes (differ by 2): ${twinPrimes.join(", ")}`)

let sum = primes.reduce((acc, cur) => acc + cur, 0)

console.log(`Sum of all primes: ${sum}`)
```

STUDENT GRADE BOOK

Manage and analyze student grades:

```
let students = [
 { name: "Alice", scores: [85, 92, 78, 90, 88] },
 { name: "Bob", scores: [72, 68, 75, 80, 77] },
 { name: "Charlie", scores: [95, 98, 92, 96, 94] },
 { name: "Diana", scores: [88, 85, 90, 87, 91] },
 { name: "Eve", scores: [60, 65, 58, 70, 62] }
]

console.log("=== GRADE BOOK ===")
console.log("")

let classAverage = 0
let topStudent = null
let topAverage = 0

for (let student of students) {
 let sum = 0

 for (let score of student.scores) {
 sum = sum + score
 }

 let average = sum / student.scores.length
 let min = Math.min(...student.scores)
 let max = Math.max(...student.scores)

 let grade
 if (average >= 90) {
 grade = "A"
 } else if (average >= 80) {
 grade = "B"
 } else if (average >= 70) {
```

```
 grade = "C"
 } else if (average >= 60) {
 grade = "D"
 } else {
 grade = "F"
 }

 console.log(`${student.name}:`)
 console.log(`  Scores: ${student.scores.join(", ")}`)
 console.log(`  Average: ${average.toFixed(1)} (Grade: ${grade})`)
 console.log(`  Range: ${min} - ${max}`)
 console.log("")

 classAverage = classAverage + average

 if (average > topAverage) {
 topAverage = average
 topStudent = student.name
 }
}

classAverage = classAverage / students.length

console.log("=== CLASS SUMMARY ===")
console.log(`Class Average: ${classAverage.toFixed(1)}`)
console.log(`Top Student: ${topStudent}
(${topAverage.toFixed(1)})`)

let passing = students.filter(s => {
 let avg = s.scores.reduce((a, b) => a + b, 0) / s.scores.length
 return avg >= 70
})

let failing = students.filter(s => {
 let avg = s.scores.reduce((a, b) => a + b, 0) / s.scores.length
 return avg < 70
})

console.log(`Passing: ${passing.map(s => s.name).join(", ")}`)
console.log(`Needs Improvement: ${failing.map(s =>
```

s.name).join(", ")}`)

PATTERN GENERATOR

Generate various patterns:

```
console.log("=== PATTERN GENERATOR ===")
console.log("")

console.log("Right Triangle:")
for (let row = 1; row <= 5; row++) {
 let line = ""
 for (let col = 1; col <= row; col++) {
 line = line + "* "
 }
 console.log(line)
}
console.log("")

console.log("Number Triangle:")
for (let row = 1; row <= 5; row++) {
 let line = ""
 for (let col = 1; col <= row; col++) {
 line = line + col + " "
 }
 console.log(line)
}
console.log("")

console.log("Pyramid:")
let height = 5
for (let row = 1; row <= height; row++) {
 let spaces = ""
 let stars = ""

 for (let s = 1; s <= height - row; s++) {
 spaces = spaces + " "
```

```
 }

 for (let c = 1; c <= 2 * row - 1; c++) {
 stars = stars + "*"
 }

 console.log(spaces + stars)
}
console.log("")

console.log("Diamond:")
for (let row = 1; row <= height; row++) {
 let spaces = ""
 let stars = ""

 for (let s = 1; s <= height - row; s++) {
 spaces = spaces + " "
 }

 for (let c = 1; c <= 2 * row - 1; c++) {
 stars = stars + "*"
 }

 console.log(spaces + stars)
}

for (let row = height - 1; row >= 1; row--) {
 let spaces = ""
 let stars = ""

 for (let s = 1; s <= height - row; s++) {
 spaces = spaces + " "
 }

 for (let c = 1; c <= 2 * row - 1; c++) {
 stars = stars + "*"
 }

 console.log(spaces + stars)
}
```

WHAT YOU BUILT

You built six complete programs demonstrating:

Number analysis with statistics
Text processing and word analysis
Shopping cart with discounts and taxes
Prime number finder with twin primes
Student grade management system
Pattern generator with multiple shapes

Each program combined:

While, for, for...of loops
Nested loops for complex patterns
Break and continue for flow control
Array methods like map, filter, reduce
Conditional logic from Book 2
All data types from Book 1

WHAT COMES NEXT

You can now make programs that:
- Make decisions (Book 2)
- Repeat operations (Book 3)

But your code is getting long. The same patterns repeat. You find yourself copying and pasting similar code.

Book 4 teaches functions. Functions let you:
- Name a block of code
- Reuse it anywhere
- Pass in different values

- Get results back

Functions are how real programs are built. They are how you organize code. They are how you stop repeating yourself.

You finished Book 3. You can make programs that loop. Now we make them modular.